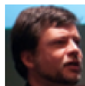
Les scripts `bash`

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

INF600A
18–25 septembre 2018

Un *tweet* de Andy Hunt, un des auteurs du (célèbre) livre «*The Pragmatic Programmer*»



Andy Hunt (@PragmaticAndy)

2016-02-11 15:10

There's few things as lovely as a short shell script that just works, and would have been much harder to do by hand.

1 Les variables
- Définition et visibilité
- Expansion des variables et *quoting*
- Les arguments d'un script
- Opérations spéciales d'expansion de variables
- Ordre des expansions

2 Les tests
- La commande `[...]`
- Les opérateurs «`&&`» et «`||`»
- Les expressions de condition
- Les tests avec `[[...]]`

3 Les opérations arithmétiques

4 Les structures de contrôle
- L'instruction `if`
- L'instruction `while`
- L'instruction `for`
- L'instruction `case`

5 Les fonctions

6 Autres éléments
- Pour déboguer des scripts
- Exécution et création de processus enfants
- Les *here documents*
- La redirection avec «`<(...)`»
- L'instruction `read`
- Suggestions pour le style et la sécurité

Remarque : Spécification du *shebang* dans les exemples

Qu'est-ce que le *shebang* ?

«Le shebang, représenté par # !, est un en-tête d'un fichier texte qui indique au système d'exploitation (de type Unix) que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.»

Source: `https://fr.wikipedia.org/wiki/Shebang`

Remarque : Spécification du *shebang* dans les exemples

- Les exemples qui suivent utilisent le *shebang* suivant :*

  #!/bin/sh -

- Sur ma machine, `sh` = `bash` :

```
$ ls -l /bin/sh
lrwxrwxrwx. 1 root root 4  3 jun  2016 /bin/sh -> bas
```

- Règle génerale (mais pas toujours !) «`#!`» = «`#!/bin/sh`».

- Certains auteurs suggèrent d'utiliser «`#!/bin/bash`»
  D'autres suggèrent plutôt «`#!/usr/bin/env bash`»

---

*. Le tiret après «`/bin/sh`» «*says that there are no more shell options ; this is a security feature to provent certain kinds of spoofing attacks.*» [RB09]

1. Les variables

1.1 Définition et visibilité

Une variable est locale au processus, à moins d'être explicitement exportée

Si exportée, elle est alors ajoutée à l'`env`ironnement

`env` donne la liste de toutes les variables connues

```
$ UNE_VAR=foo

$ echo $UNE_VAR
foo

$ env | grep UNE_VAR

$ export UNE_VAR

$ env | grep UNE_VAR
UNE_VAR=foo
```

Une variable peut être supprimée de l'environnement avec `unset`

Une variable non définie équivaut à une chaine vide

```
$ env | grep UNE_VAR
UNE_VAR=foo

$ unset UNE_VAR

$ env | grep UNE_VAR

$ echo $UNE_VAR

$ echo "'$UNE_VAR'"
''
```

Une variable est locale au processus, à moins d'être explicitement exportée

`export -p` émet les variables exportées sur `stdout`

```
$ unset UNE_VAR

$ UNE_VAR=foo

$ echo $UNE_VAR
foo

$ export -p | grep UNE_VAR

$ export UNE_VAR

$ export -p | grep UNE_VAR
declare -x UNE_VAR="foo"
```

La primitive `export` rend la variable visible dans tous les processus enfants


```
$ cat verifier-var.sh
#!/bin/sh -
export -p | grep UNE_VAR

$ UNE_VAR=foo

$ ./verifier-var.sh

# Devient visible apres le export.
$ export UNE_VAR=foo

$ ./verifier-var.sh
declare -x UNE_VAR="foo"
```

On peut aussi rendre visible une variable dans un seul processus enfant


```
$ cat verifier-var.sh
#!/bin/sh -
export -p | grep UNE_VAR

$ unset UNE_VAR

$ ./verifier-var.sh

# Visible uniquement pour ce processus.
$ UNE_VAR=foo ./verifier-var.sh
declare -x UNE_VAR="foo"

$ ./verifier-var.sh
```

1.2 Expansion des variables et *quoting*

Le «\» supprime l'intepprétation spéciale du caractère qui suit.

```
$ x=100

$ echo $x
100

$ echo \$x
$x

$ echo &
[1] 14641

[1]+  Fini                    echo

$ echo \&
&
```

Il existe trois façons de supprimer l'interprétation spéciale de certains caractères spéciaux

2. Avec les apostrophes simples

Les apostrophes (guillemets simples) conservent de façon littérale **tout** ce qui est indiqué entre les apostrophes.

```
$ x=100

$ echo $x""$x
100100

$ echo '$x""$x'
$x""$x

$ ls
fich0.txt  fich1.txt

$ echo *
fich0.txt fich1.txt

$ echo '*'
*
```

Les guillemets (doubles) conservent de façon littérale ce qui est entre les guillemets, **sauf pour «$» et «\»**.

```
$ x=100

$ echo "$x''$x"
100''100

$ echo "$x""$x"
100100

$ echo "$x\"\"$x"
100""100

$ echo "\$x\$x"
$x$x
```

Donc l’expansion de variable se fait dans les guillemets ("..."), mais pas dans les apostrophes ('...')


```
$ AUTRE_VAR=$HOME:$UNE_VAR--$EDITOR

$ echo $AUTRE_VAR
/home/tremblay:foo--emacs

$ AUTRE_VAR="$HOME $UNE_VAR $EDITOR"

$ echo $AUTRE_VAR
/home/tremblay foo emacs


$ AUTRE_VAR='$HOME $UNE_VAR $EDITOR'

$ echo $AUTRE_VAR
??

$ AUTRE_VAR=$HOME $UNE_VAR $EDITOR
??
```

Donc l'expansion de variable se fait dans les guillemets ("..") , mais pas dans les apostrophes ('..')


```
$ AUTRE_VAR=$HOME:$UNE_VAR--$EDITOR

$ echo $AUTRE_VAR
/home/tremblay:foo--emacs

$ AUTRE_VAR="$HOME $UNE_VAR $EDITOR"

$ echo $AUTRE_VAR
/home/tremblay foo emacs


$ AUTRE_VAR='$HOME $UNE_VAR $EDITOR'

$ echo $AUTRE_VAR
$HOME $UNE_VAR $EDITOR

$ AUTRE_VAR=$HOME $UNE_VAR $EDITOR
-bash: foo : commande introuvable
```

L'expansion des variables se fait aussi dans le nom de la commande, pas juste dans les arguments

Les accolades {...} peuvent être utilisées pour délimiter un nom de variable

```
$ LS="ls -l"

$ F=foo

$ $LS $F.*
-rwxr-xr-x 1 tremblay [...] foo.sh


$ F=foo.s

$ $LS $Fh
??

$ $LS ${F}h
??
```

L’expansion des variables se fait aussi dans le nom de la commande, pas juste dans les arguments

Les accolades {...} peuvent être utilisées pour délimiter un nom de variable

```
$ LS="ls -l"

$ F=foo

$ $LS $F.*
-rwxr-xr-x 1 tremblay [...] foo.sh


$ F=foo.s

$ $LS $Fh
-rwxr-xr-x. 1 tremblay [...] args2.sh
...
-rwxr-xr-x. 1 tremblay [...] verifier-var.sh

$ $LS ${F}h
-rwxr-xr-x 1 tremblay [...] foo.sh
```

La «substitution de commandes» exécute une commande et insére le résultat comme liste de mots

Forme moderne recommandée : $(...)

```
$ cat foo1.txt
abc
def

$ cat foo2.txt
123
456
```

```
$ echo "ls -1 foo*.txt"
ls -1 foo*.txt

$ ls -1 foo*.txt
foo1.txt
foo2.txt
```

```
$ echo ls -1 foo*.txt
??
```

```
$ echo $(ls -1 foo*.txt)
??
```

La «substitution de commandes» exécute une commande et insére le résultat comme liste de mots

Forme moderne recommandée : $(. . .)

```
$ cat foo1.txt
abc
def

$ cat foo2.txt
123
456
```

```
$ echo "ls -1 foo*.txt"
ls -1 foo*.txt

$ ls -1 foo*.txt
foo1.txt
foo2.txt
```

```
$ echo ls -1 foo*.txt
ls -1 foo1.txt foo2.txt
```

```
$ echo $(ls -1 foo*.txt)
foo1.txt foo2.txt
```

La «substitution de commandes» exécute une commande et insére le résultat comme liste de mots

Forme moderne... et **recommandée** : $(...)

```
$ cat foo1.txt
abc
def

$ cat foo2.txt
123
456
```

```
$ cat ls foo*.txt
??
```

```
$ cat $(ls foo*.txt)
??
```

La «substitution de commandes» exécute une commande et insére le résultat comme liste de mots

Forme moderne… et **recommandée** : $(…)

```
$ cat foo1.txt
abc
def

$ cat foo2.txt
123
456
```

```
$ cat ls foo*.txt
cat: ls: No such file or directory
abc
def
123
456
```

```
$ cat $(ls foo*.txt)
abc
def
123
456
```

Forme ancienne

```
$ cat foo1.txt
abc
def

$ cat foo2.txt
123
456
```

```
$ echo "ls foo*.txt"
ls foo*.txt

$ echo ls foo*.txt
ls foo1.txt foo2.txt
```

```
$ cat ls foo*.txt
cat: ls: No such file or directory
abc
def
123
456

$ echo `ls foo*.txt`
foo1.txt foo2.txt

$ cat `ls foo*.txt`
abc
def
123
456
```

La substitution des commandes effectue aussi l'expansion des variables

Mais l'expansion des noms de fichiers ne se fait pas à l'intérieur de guillemets

```
$ ls f*.txt
f0.txt f1.txt f2.txt f3.txt ff0.txt ff.txt f.txt

$ LS="ls f*.txt"

$ echo "'$LS'"
??

$ echo '"$LS"'
??

$ $LS
??

$ echo $($LS)
??

$ $($LS)
??
```

La substitution des commandes effectue aussi l’expansion des variables

Mais l’expansion des noms de fichiers ne se fait pas à l’intérieur de guillemets

```
$ ls f*.txt
f0.txt f1.txt f2.txt f3.txt ff0.txt ff.txt f.txt

$ LS="ls f*.txt"

$ echo "'$LS'"
'ls f*.txt'

$ echo '"$LS"'
"$LS"

$ $LS
f0.txt f1.txt f2.txt f3.txt ff0.txt ff.txt f.txt

$ echo $($LS)
f0.txt f1.txt f2.txt f3.txt ff0.txt ff.txt f.txt

$ $($LS)
-bash: ./f0.txt: Permission non accordee
```

Une variable peut être définie par une substitution de commande


```
$ LS=$(ls f*.txt)

$ echo $LS
f0.txt f1.txt f2.txt f3.txt ff0.txt ff.txt f.txt

$ cat $LS
bc
def
ghi
.
.
.
a+ b cd
a* b* c*d*
aaa bbb ccc
```

1.3 Les arguments d’un script

Les arguments sont numérotés à partir de `$1`

L'argument $0 est le nom du script (nom du fichier)

```
$ cat args.sh
#!/bin/sh -
echo "Nb. arguments = $#"
echo "  \$0 = '$0'"
echo "  \$1 = '$1'"
echo "  \$2 = '$2'"

$ ./args.sh
Nb. arguments = 0
  $0 = './args.sh'
  $1 = ??
  $2 = ??

$ ./args.sh 99 1234 344 34
Nb. arguments = 4
  $0 = './args.sh'
  $1 = '99'
  $2 = '1234'
```

Les arguments sont numérotés à partir de `$1`

L'argument $0 est le nom du script (nom du fichier)

```
$ cat args.sh
#!/bin/sh -
echo "Nb. arguments = $#"
echo "  \$0 = '$0'"
echo "  \$1 = '$1'"
echo "  \$2 = '$2'"

$ ./args.sh
Nb. arguments = 0
  $0 = './args.sh'
  $1 = ''
  $2 = ''

$ ./args.sh 99 1234 344 34
Nb. arguments = 4
  $0 = './args.sh'
  $1 = '99'
  $2 = '1234'
```

On procède à l’expansion des variables dans les arguments, en considérant les caractères *escapés*


```
$ X=88

$ Y=Z

$ ./args.sh "$X $Y" 10\ 30 90
Nb. arguments = 3
  $0 = './args.sh'
  $1 = '88 Z'
  $2 = '10 30'
```

La commande `shift` «décale vers la gauche» les arguments, mais sans toucher à $0

Utile pour traiter chacun des arguments

Soit un script `foo.sh` appelé comme suit :

```
$ ./foo.sh 10 20 30 40 50
```

Donc on a les arguments comme suit (avec `$# == 5`) :

$0	$1	$2	$3	$4	$5
./foo.sh	10	20	30	40	50

Après un premier appel à `shift` (`$# == 4`) :

$0	$1	$2	$3	$4
./foo.sh	20	30	40	50

Après un autre appel à `shift` (`$# == 3`) :

$0	$1	$2	$3
./foo.sh	30	40	50

La commande `shift` est utile pour traiter chacun des arguments, un après l'autre


```
$ cat args2.sh
#!/bin/sh -
echo "Nb. arguments = $#"
echo "  \$0 = '$0'"
echo "  \$1 = '$1'"
shift
echo "  \$2 = '$1'"
shift
echo "  \$3 = '$1'"

$ args2.sh 10 20 30 40 50
Nb. arguments = 5
  $0 = './args2.sh'
  $1 = '10'
  $2 = '20'
  $3 = '30'
```

```
$ ls f*
f0.txt  f1.txt  f2.txt  f3.txt  ff0.txt  ff.txt
foo.sh*  f.txt

$ ls -1 f* | wc -l
8

$ ./args.sh f*
Nb. arguments = ??
  $0 = './args.sh'
  $1 = ??
  $2 = ??

$ ./args.sh  "f*"
Nb. arguments = ??
  $0 = './args.sh'
  $1 = ??
  $2 = ??
```

L’expansion des noms de fichier (*file globbing*) ne se fait pas à l’intérieur des guillemets


```
$ ls f*
f0.txt  f1.txt  f2.txt  f3.txt  ff0.txt  ff.txt
foo.sh*  f.txt

$ ls -1 f* | wc -l
8

$ ./args.sh f*
Nb. arguments = 8
  $0 = './args.sh'
  $1 = 'f0.txt'
  $2 = 'f1.txt'

$ ./args.sh  "f*"
Nb. arguments = 1
  $0 = './args.sh'
  $1 = 'f*'
  $2 = ''
```

La variable «$@» donne accès à la liste de tous les arguments

Mais attention : il faut mettre cette variable entre "..."

```
$ cat args.sh
#!/bin/sh -
for x in "$@"
do
    echo "$x"
done
```

```
$ ./args.sh "ab cd" 12\ 34 ZZZ
ab cd
12 34
ZZZ
```

La variable «$@» donne accès à la liste de tous les arguments

Mais attention : il faut mettre cette variable entre "..."

```
$ cat args.sh
#!/bin/sh -
for x in $@
do
    echo "$x"
done
```

```
$ ./args.sh "ab cd" 12\ 34 ZZZ
??
```

La variable «`$@`» donne accès à la liste de tous les arguments

Mais attention : il faut mettre cette variable entre "..."

```
$ cat args.sh
#!/bin/sh -
for x in $@
do
    echo "$x"
done
```

```
$ ./args.sh "ab cd" 12\ 34 ZZZ
ab
cd
12
34
ZZZ
```

Équivalences :

- `"$@" == "$1" "$2" "$3" ...`
- `$@ == $1 $2 $3 ...`

- Presque toujours (!) préférable d'utiliser "`$@`".

La variable «$*» donne aussi accès à la liste de tous les arguments

Tous en une seule et unique chaine lorsqu'entre guillemets

```
$ cat args.sh
#!/bin/sh -
for x in "$*"
do
    echo "$x"
done
```

```
$ ./args.sh "ab cd" 12\ 34 ZZZ
ab cd 12 34 ZZZ
```

La variable «$*» donne aussi accès à la liste de tous les arguments

Mais a le même comportement que $@ lorsque sans guillemet

```
$ cat args.sh
#!/bin/sh -
for x in $*
do
    echo "$x"
done
```

```
$ ./args.sh "ab cd" 12\ 34 ZZZ
??
```

La variable «$*» donne aussi accès à la liste de tous les arguments

Mais a le même comportement que $@ lorsque sans guillemet

```
$ cat args.sh
#!/bin/sh -
for x in $*
do
    echo "$x"
done
```

```
$ ./args.sh "ab cd" 12\ 34 ZZZ
ab
cd
12
34
ZZZ
```

Équivalences :

- `"$*" == "$1 $2 $3 ..."`
- `$* == $1 $2 $3 ...`

- Généralement préférable d'utiliser `"$@"` !

1.4 Opérations spéciales d’expansion de variables

L'utilisation de `${...}` permet de séparer un nom de variable des caractères qui suivent


```
$ X=abc

$ echo ${X}
abc

$ echo $XY

$ echo ${X}Y
abcY
```

L'utilisation de `${...}` permet de spécifier une valeur par défaut lorsqu'une variable est vide

Sans ou avec affectation à la variable

Retour d'une valeur par défaut si non définie (sans modification)

```
$ unset X

$ echo $X

$ echo ${X:-xyz}
xyz

$ echo $X

$
```

Affectation d'une valeur si non définie (avec modification)

```
$ unset X

$ echo $X

$ echo ${X:=xyz}
xyz

$ echo $X
xyz

$
```

On peut faire certaines opérations simples de *pattern matching* avec `${...}`

Les motifs sont ceux du *file globbing*, et non pas ceux des expressions régulières

```
$ X=abcdefabc

$ echo ${X#*a}                                    # ??
bcdefabc

$ echo $X
abcdefabc

$ echo ${X##*a}                                   # ??
bc
```

- «#» : Trouve la sous-chaine la plus courte qui matche à partir du début et supprime cette sous-chaine !
- «##» : Trouve la sous-chaine la plus longue qui matche à partir du début et supprime cette sous-chaine !

Note : Retourne une chaine résultat sans modifier la variable initiale !

On peut faire certaines opérations simples de *pattern matching* avec ${...}

Les motifs sont ceux du *file globbing*, et non pas ceux des expressions régulières

```
$ X=abcdefabc

$ echo ${X#*a}                                   # [a]bcdefabc
bcdefabc

$ echo $X
abcdefabc

$ echo ${X##*a}                                  # [abcdefa]bc
bc
```

- «#» : Trouve la sous-chaine la plus courte qui matche à partir du début et supprime cette sous-chaine !
- «##» : Trouve la sous-chaine la plus longue qui matche à partir du début et supprime cette sous-chaine !

Note : Retourne une chaine résultat sans modifier la variable initiale !

On peut aussi faire certaines opérations de *pattern matching* avec `${...}`

Les motifs sont ceux du *file globbing*, et non pas les expressions régulières

```
$ X=abcdefabc


$ echo ${X%b*}                                          # ??
abcdefa


$ echo ${X%%b*}                                         # ??
a
```

- «%» : Trouve la sous-chaine la plus courte qui matche à partir de la fin et supprime cette sous-chaine !
- «%%» : Trouve la sous-chaine la plus longue qui matche à partir de la fin et supprime cette sous-chaine !

On peut aussi faire certaines opérations de *pattern matching* avec `${...}`

Les motifs sont ceux du *file globbing*, et non pas les expressions régulières

```
$ X=abcdefabc

$ echo ${X%b*}                              # abcdefa[bc]
abcdefa

$ echo ${X%%b*}                             # a[bcdefabc]
a
```

- «%» : Trouve la sous-chaine la plus courte qui matche à partir de la fin et supprime cette sous-chaine !
- «%%» : Trouve la sous-chaine la plus longue qui matche à partir de la fin et supprime cette sous-chaine !

Truc mnémonique pour l'interprétation de ces symboles

- On écrit «# 1» donc # est devant $\Rightarrow$ début
- On écrit «100 %» donc % est après $\Rightarrow$ fin
- court :
  «#» ou «%»

  long :
  «##» ou «%%»

On peut utiliser le *pattern matching* avec `${...}` pour obtenir les éléments d'un nom de fichier


```
$ fich=/home/tremblay/cmd/mk-transparents.sh

$ echo ${fich%/*}
/home/tremblay/cmd

$ nomDeBase=${fich##*/}
$ echo $nomDeBase
mk-transparents.sh

$ echo ${nomDeBase%%.*}
mk-transparents

$ echo ${fich##*.}
sh
```

On peut utiliser le *pattern matching* avec `${...}` pour obtenir les éléments d'un nom de fichier


```
$ fich=/home/tremblay/cmd/mk-transparents.sh

$ echo ${fich%/*}
/home/tremblay/cmd

$ nomDeBase=${fich##*/}
$ echo $nomDeBase
mk-transparents.sh

$ echo ${nomDeBase%%.*}
mk-transparents

$ echo ${fich##*.}
sh
```

Remarque : Il existe une commande `basename`

```
$ basename $fich
mk-transparents.sh
```

On peut utiliser le *pattern matching* avec `${...}` pour obtenir les éléments d'un nom de fichier

Mais on doit faire des tests additionnels pour les chemins incomplets

```
$ fich=foo

$ echo ${fich##*/}
foo

$ echo ${fich##*.}
foo

$ [[ ${fich##*.} == $fich ]] && echo "Pas d'extension"
Pas d'extension

$ echo ${fich%/*}
foo

$ [[ ${fich%/*} == $fich ]] && echo "Pas de prefixe"
Pas de prefixe
```

1.5 Ordre des expansions

Ordre dans lequel se font les diverses expansions et évaluations ⋆

1. Accolades dans les noms de fichiers (e.g., `abc{1,2,8}.txt`)
2. `~` ⇒ répertoire `home`
3. Variables et paramètres — `$nomVar`, `${nomVar}`, `$0`, etc.
4. Substitution de commandes — `$( ... )`
5. Opérations arithmétiques — `(( ... ))`
6. Décomposition en mots distincts — séparateur (défaut) = blancs
7. Noms de fichiers (*file globbing*) — `* ? [...] [!...]`

Puis, on «exécute» la liste des mots résultants — où le 1er mot est la commande

Pour plus de détails : `http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_03_04.html`

Ordre dans lequel se font les diverses expansions et évaluations

Exemple

```
$ pwd
/home/tremblay/Tmp

$ cat foo.txt
echo
cat

$ x=foo

$ $(grep h ~/Tmp/$x.txt) foo*
foo.txt

$ $(grep a $x.txt) foo*
echo
cat
```

2. Les tests

2.1 La commande [. . .]

La commande `test` évalue une condition et retourne un *exit status* — avec Vrai/OK = 0 !


```
$ type test
test est une primitive du shell

$ X=10

$ test X = 10; echo $?
1

$ test $X = 10; echo $?
0

$ test $X = 29; echo $?
1
```

Le test `[ ... ]` est un synonyme de la commande `test`


```
$ type [
[ est une primitive du shell

$ X=10

$ [ X = 10 ]; echo $?
1

$ [ $X = 10 ]; echo $?
0

$ [ $X = 29 ]; echo $?
1


$ [$X = FOO]
??
```

Le test [...] est un synonyme de la commande test


```
$ type [
[ est une primitive du shell

$ X=10

$ [ X = 10 ]; echo $?
1

$ [ $X = 10 ]; echo $?
0

$ [ $X = 29 ]; echo $?
1


$ [$X = FOO]
-bash: [10 : commande introuvable
```

Le test `[...]` est un synonyme de la commande `test`

[is a command, not a syntax marker for the `if` *statement. It's equivalent to the* `test` *command, except that the final argument must be a].*

Source: `https://mywiki.wooledge.org/BashPitfalls`

2.2 Les opérateurs « && » et « || »

Le test « [expr] » peut s'utiliser avec &&


```
$ X=10

$ [ $X = 10 ] && echo "Egal a 10"
Egal a 10

$ [ $X = 99 ] && echo "Egal a 10"

$
```

Le test « `[ expr ]` » peut s'utiliser avec `&&`


```
$ X=10

$ [ $X = 10 ] && echo "Egal a 10"
Egal a 10

$ [ $X = 99 ] && echo "Egal a 10"

$
```

Note : Dans un tel cas, le test combiné avec le `&&` peut être vu comme une forme de garde — une condition qui doit être vraie pour que l'action qui suit s'exécute !

En Ruby, on écrirait :

```
puts "Egal a 10" if x == 10
```

Le test « `[ expr ]` » peut s'utiliser avec `||`


```
$ X=10

$ [ $X == 10 ] || echo "Pas 10"

$ [ $X == 0 ] || echo "Pas 0"
Pas 0

$ [ $X != 10 ] || echo "Egal a 10"
Egal a 10

$ [ $X != 0 ] || echo "Est 0"

$
```

```
$ X=10

$ [ $X == 10 ] || echo "Pas 10"

$ [ $X == 0 ] || echo "Pas 0"
Pas 0

$ [ $X != 10 ] || echo "Egal a 10"
Egal a 10

$ [ $X != 0 ] || echo "Est 0"

$
```

Note : Ici, le test combiné avec `||` est une forme de garde négative — une condition qui doit être fausse pour que l'action qui suit s'exécute !

En Ruby, on écrirait (1*er* exemple) (`unless cond == if !cond`) :

```
puts "Pas 10" unless x == 10
```

Les opérateurs `&&` et `||` s’utilisent avec n’importe quelle commande et utilisent l’*exit status*


```
$ ls machin* &> /dev/null || echo "Aucun fichier machin*
Aucun fichier machin*


$ ls foo* &> /dev/null && echo "Il y a des fichiers foo*
Il y a des fichiers foo*
```

Les opérateurs `&&` et `||` s'utilisent avec n'importe quelle commande et utilisent l'*exit status* ⋆


```
$ cat ok.sh
#!/bin/sh -
exit 0

$ cat pas-ok.sh
#!/bin/sh -
exit 1
```

```
$ ./ok.sh && echo "OK"
OK

$ ./pas-ok.sh && echo "OK"

$ ./ok.sh || echo "PAS OK"

$ ./pas-ok.sh || echo "PAS OK"
PAS OK
```

2.3 Les expressions de condition

Différentes formes d'expressions peuvent être utilisées dans une commande `test/[]`

Propriétés de fichiers

Expression	**Vrai si...**
`-d fichier`	`fichier` est un répertoire
`-e fichier`	`fichier` existe
`-f fichier`	`fichier` est un fichier ordinaire
`-h fichier`	`fichier` est lien symbolique
`-s fichier`	`fichier` n'est pas vide
`-x fichier`	`fichier` est exécutable
...	...

Différentes formes d'expressions peuvent être utilisées dans une commande `test/[]`

Propriétés de chaines

Expression	**Vrai si...**
`chaine`	`chaine` n'est pas vide
`-z chaine`	`chaine` est vide
`ch1 = ch2`	Les chaines sont identiques
`ch1 != ch2`	Les chaines sont différentes

Différentes formes d’expressions peuvent être utilisées dans une commande `test/[]`

Propriétés numériques

Expression	Vrai si...
`n1 -eq n2`	`n1` == `n2`
`n1 -ne n2`	`n1` != `n2`
`n1 -lt n2`	`n1` < `n2`
`n1 -gt n2`	`n1` > `n2`
`n1 -le n2`	`n1` <= `n2`
`n1 -ge n2`	`n1` >= `n2`

Différentes formes d'expressions peuvent être utilisées dans une commande `test/[]`

Propriétés numériques

Expression	Vrai si...
`n1 -eq n2`	n1 == n2
`n1 -ne n2`	n1 != n2
`n1 -lt n2`	n1 < n2
`n1 -gt n2`	n1 > n2
`n1 -le n2`	n1 <= n2
`n1 -ge n2`	n1 >= n2

Donc, «=» = comparaison de chaines vs. «-eq» = comparaison numérique

```
$ x=01

$ [ $x = 1 ] && echo "OUI"

$ [ $x -eq 1 ] && echo "OUI"
OUI
```

2.4 Les tests avec `[[ ... ]]`

Les tests peuvent aussi être écrits avec `[[...]]`

Forme plus «récente», disponible en `bash` et d'autres *shells*, mais peut-être pas dans certains vieux *shells*

```
$ ls -l f1.txt
-rw-r--r--  1 tremblay  staff
12 Jan 14 14:50 f1.txt

$ [[ -e f1.txt ]] && echo "f1.txt existe"
f1.txt existe


$ FICH="un fichier.txt"

$ [[ -f $FICH ]] && echo "$FICH existe"
un fichier.txt existe
```

Note : Alors que « `[]` » est une commande primitive (*builtin*), « `[[   ]]` » est un élément de syntaxe du *shell*. †

†. Introduit par `ksh` puis repris par `bash`.

Un avantage des tests avec `[[...]]` : Certains guillemets peuvent être omis


```
$ FICH="un fichier.txt"


$ [ -f "$FICH" ] && echo "$FICH existe"
un fichier.txt existe


$ [ -f $FICH ] && echo "$FICH existe"
bash: [: un: binary operator expected


$ [[ -f $FICH ]] && echo "$FICH existe"
un fichier.txt existe
```

Note : L'utilisation de « `[[  ]]` » prévient de nombreuses erreurs, donc c'est maintenant la forme généralement recommandée.

Un avantage des tests avec `[[...]]` : Certains *backslashes* peuvent être omis


```
$ [[ abc < def ]] && echo "OK"
??

$ [ abc < def ] && echo "OK"
??

$ [ abc \< def ] && echo "OK"
??

$ echo $abc","$def
9,444

$ [[ $abc < $def ]] && echo "OK"
??

$
```

??

Un avantage des tests avec `[[...]]` : Certains *backslashes* peuvent être omis


```
$ [[ abc < def ]] && echo "OK"
OK

$ [ abc < def ] && echo "OK"
bash: def: No such file or directory

$ [ abc \< def ] && echo "OK"
OK

$ echo $abc","$def
9,444

$ [[ $abc < $def ]] && echo "OK"


$
```

Donc «<>» reste une comparaison alphabétique par défaut, et non une comparaison numérique !

Le comportement de « [. . .] » est comme n'importe quelle autre commande, mais pas celui de [[. . .]] ★[63]

```
$ FICH="un fichier.txt"

$ ls -l "$FICH"
-rw-r--r--  1 tremblay  staff
0 Jan 22 15:14 un fichier.txt

$ ls -l $FICH
ls: fichier.txt: No such file or directory
ls: un: No such file or directory


$ [[ -f $FICH ]] && echo "$FICH existe"
un fichier.txt existe
```

Un autre avantage de `[[...]]` : Permet le *pattern-matching* (expr. rég. étendues)


```
$ FICH="un fichier.txt"

$ [[ $FICH =~ ^un.*\ .*t$ ]] && echo "Matche"
Matche

$ [[ $FICH =~ ^[a-z]+\ [a-z]+.txt$ ]] && echo "Matche"
Matche

$ [[ $FICH =~ ^[^0-9]$ ]] && echo "Matche"

$ [[ $FICH =~ ^[^0-9]+$ ]] && echo "Matche"
Matche
```

Un autre avantage de `[[...]]` : Permet des expressions complexes avec `&&` et `||`


```
$ x=abc; y=def

$ [[ $x == "abc" && $y == "def" ]] && echo OUI
OUI

$ [[ $x == "def" || $y == "def" ]] && echo OUI
OUI


$ [ $x == "abc" && $y == "def" ] && echo OUI
-bash: [: << ] >> manquant

$ [ $x == "abc" \&\& $y == "def" ] && echo OUI
-bash: [: trop d'arguments
```

3. Les opérations arithmétiques

Par défaut, toutes les variables sont des chaines


```
$ x=10

$ x=$x+1

$ echo $x
??

$ x=$x+1

$ echo $x
??
```

Par défaut, toutes les variables sont des chaines


```
$ x=10

$ x=$x+1

$ echo $x
10+1

$ x=$x+1

$ echo $x
10+1+1
```

Par défaut, toutes les variables sont des chaines (bis)

Sur mes machines personnelles…

```
$ x=10

$ x=$x + 1
bash: pushd: 1: No such file or directory

$ which +

+ is aliased to ‘pushd+’
```

On interprète des instructions comme étant des opérations arithmétiques avec `((...))`

Indulgent quant à la forme des expressions

```
$ x=10

$ (( x=$x+1 ))

$ echo $x
11

$ (( x=x+1 ))

$ echo $x
12

$ (( x = x+1 ))

$ echo $x
13
```

4. Les structures de contrôle

4.1 L’instruction `if`

Forme générale d'une instruction `if`

Une *condition* est n'importe quelle commande qui retourne un *exit status*

```
if condition
then
  instruction
  ...
elif condition
then
 instruction
 ...
.
.
.
else
 instruction
 ...
fi
```

Une commande peut être utilisée comme condition

Puisqu'une commande retourne toujours un code de statut (*exit status*)

```
$ cat f1.txt
abc
def
ghi

$ if grep -q abc f1.txt; then
>   echo "f1.txt contient abc"
> fi
f1.txt contient abc
```

Une commande peut être utilisée comme condition

Puisqu'une commande retourne toujours un code de statut (*exit status*)

La commande `ls`... est trop bavarde !

```
$ if ! ls machin.*; then
>   echo "machin.*" existe pas
> fi
ls: impossible d'accéder à machin.*: Aucun fichier ou do
machin.* existe pas
```

On peut ignorer son résultat en l'envoyant de la poubelle à bits (*bit bucket*)

```
$ if ! ls machin.* &> /dev/null
> then
>   echo "machin.*" existe pas
> fi
machin.* existe pas
```

La position du `then` est importante

On doit mettre « ;» si on met le `then` après la condition. . . et c’est souvent ce qu’on fait

```
$ cat if3.sh
#!/bin/sh -

if [ -f f.txt ]
then
  echo "1. f.txt existe"
fi

if [ -f f.txt ]; then
  echo "2. f.txt existe"
fi

if [ -f f.txt ] then
  echo "3. f.txt existe"
fi
```

```
$ ./if3.sh
1. f.txt existe
2. f.txt existe
./if3.sh: line 14: Erreur de\
   syntaxe pres du symbole\
   inattendu " fi "
./if3.sh: line 14: `fi'
```

On peut avoir plusieurs clauses `elif`


```
$ cat if4.sh
#!/bin/sh -

if [ -f foo.abc ]; then
  echo "foo.abc existe"
elif [ -f foo.txt ]; then
  echo "foo.txt existe"
elif [ -f foo.sh ]; then
  echo "foo.sh existe"
else
  echo "foo.{abc,txt,sh} existent pas"
fi
```

4.2 L’instruction `while`

L’instruction `while` est utilisée pour les boucles indéfinies (nombre variable d’itérations)

Les opérations se font par défaut… sur des chaines !

```
$ cat args3.sh
#!/bin/sh -
echo "Nb. arguments = $#"

i=0
while [ $# != 0 ]
do
    i=$i+1
    echo "  \$$i = '$1'"
    shift
done

$ args3.sh 10 20 30
Nb. arguments = 3
  ??
```

L'instruction `while` est utilisée pour les boucles indéfinies (nombre variable d'itérations)

Les opérations se font par défaut… sur des chaines !

```
$ cat args3.sh
#!/bin/sh -
echo "Nb. arguments = $#"

i=0
while [ $# != 0 ]
do
    i=$i+1
    echo "  \$$i = '$1'"
    shift
done

$ args3.sh 10 20 30
Nb. arguments = 3
  $0+1 = '10'
  $0+1+1 = '20'
  $0+1+1+1 = '30'
```

L'instruction `while` est utilisée pour les boucles indéfinies (nombre variable d'itérations)

Pour les opérations arithmétiques, il faut utiliser `((...))`

```
$ cat args4.sh
#!/bin/sh -
echo "Nb. arguments = $#"

i=0
while [ $# != 0 ]
do
    (( i = $i+1 ))
    echo "  \$$i = '$1'"
    shift
done

$ args4.sh 10 20 30
Nb. arguments = 3
  $1 = '10'
  $2 = '20'
  $3 = '30'
```

L'instruction `while` est utilisée pour les boucles indéfinies (nombre variable d'itérations)

Pour les opérations arithmétiques, il faut utiliser `((...))`

```
$ cat args4.sh
#!/bin/sh -
echo "Nb. arguments = $#"

i=0
while [ $# != 0 ]
do
    (( i = i+1 ))
    echo "  \$$i = '$1'"
    shift
done

$ args4.sh 10 20 30
Nb. arguments = 3
  $1 = '10'
  $2 = '20'
  $3 = '30'
```

4.3 L'instruction `for`

L'instruction `for` est utilisée pour les boucles définies, notamment avec des listes d'éléments


```
$ type for1
for1 est une fonction
for1 ()
{
    for x in abc 123 "xyz abc";
    do
        echo $x;
    done
}
```

```
$ for1
abc
123
xyz abc
```

L'instruction `for` est utilisée pour les boucles définies, notamment avec des listes d'éléments


```
$ ls f*.txt
f0.txt f1.txt f2.txt

$ cat for2.sh
#!/bin/sh -
for x in ls f?.txt
do
  echo $x
done

echo "***"

for x in $(ls f?.txt)
do
  echo $x
done
```

```
$ ./for2.sh
??
***
??
```

L'instruction `for` est utilisée pour les boucles définies, notamment avec des listes d'éléments


```
$ ls f*.txt
f0.txt f1.txt f2.txt

$ cat for2.sh
#!/bin/sh -
for x in ls f?.txt
do
  echo $x
done

echo "***"

for x in $(ls f?.txt)
do
  echo $x
done
```

```
$ ./for2.sh
ls
f0.txt
f1.txt
f2.txt
***
f0.txt
f1.txt
f2.txt
```

Une instruction `for` style C avec opérations arithmétiques peut aussi être utilisée


```
$ for (( i = 0; i < 3; i++ )); do
> echo $i
> done
0
1
2
```

Une instruction `for` style C avec opérations arithmétiques peut aussi être utilisée ⋆

```
$ cat args_for.sh
#!/bin/sh -
echo "Nb. arguments = $#"

for (( i=0; i<=$#; i++ ))
do
  echo "  \$$i = $i"
done
```

```
$ ./args_for.sh 23 abc def 99
Nb. arguments = 4
  $0 = 0
  $1 = 1
  $2 = 2
  $3 = 3
  $4 = 4
```

Une instruction `for` style C avec opérations arithmétiques peut aussi être utilisée

Mais pour référer aux arguments, on doit utiliser l'opérateur d'indirection

```
$ cat args_for.sh
#!/bin/sh -
echo "Nb. arguments = $#"

for (( i=0; i<=$#; i++ ))
do
  echo "  \$$i = ${!i}"
done
```

```
$ ./args_for.sh 23 abc def 99
Nb. arguments = 4
  $0 = ./args_for.sh
  $1 = 23
  $2 = abc
  $3 = def
  $4 = 99
```

4.4 L'instruction `case`

Les motifs d'une instruction `case` sont ceux de l'expansion des noms de fichier (*file globbing*)

Donc, pas des expressions régulières comme `sed`, `grep` ou `awk`

La fin du motif pour un cas est indiquée par «) » et la fin d'un cas par « ;; »

```
$ cat cm.sh
#!/bin/sh -

chaine="$1"
motif="$2"

case "$chaine" in
    $motif) echo "Match";;
esac
```

```
$ ./cm.sh "abc" "ab"

$ ./cm.sh "abc" "ab*"
Match


$ ./cm.sh "abc" "^abc"

$ ./cm.sh "^abc" "^abc"
Match
```

Un cas est terminé par « ;; » et le dernier cas est généralement « *) » (cas par défaut)

Un script pour déterminer le type d'un fichier : Le script

```
$ cat case1.sh
#!/bin/sh -
option="$1"
fichier="$2"

case $option in
    -d) [ -d $fichier ] &&
        echo "Repertoire: '$fichier'";;
    -f) [ -f $fichier ] &&
        echo "Fichier: '$fichier'";;
    -x) [ -x $fichier ] &&
        echo "Executable: '$fichier'";;
    [*])
        echo "Un *";;
    *)
        echo "Option non traitee"
        exit 1;;
esac
```

Un cas est terminé par « ;; » et le dernier cas est généralement « *) » (cas par défaut)

Un script pour déterminer le type d'un fichier : Exemples d'appels

```
$ ./case1.sh -d .
Repertoire: '.'

$ ./case1.sh -x case1.sh
Executable: 'case1.sh'

$ ./case1.sh \*
Un *

$ ./case1.sh -a; echo $?
Option non traitee
1
```

Les valeurs `true` et `false` sont des primitives qui retournent un code de statut (0 ou 1) ⋆


```
$ cat case2.sh
#!/bin/sh -
case $# in
    1) echo "Un argument";;
    *) exit 1;;
esac

case $1 in
    *[!0-9]*) false;;
    *) echo "Un nombre";;
esac
```

```
$ ./case2.sh 10
Un argument
Un nombre

$ ./case2.sh; echo $?
1

$ ./case2.sh xyz; echo $?
Un argument
1
```

5. Les fonctions

Une fonction s'exécute pour ses effets et pour le *exit status* qu'elle retourne ??

Plus efficace qu'une commande : conservée en mémoire et ne crée pas de processus

```
$ cat eun.sh
#!/bin/sh -
est_un_nombre() {
  if [[ $1 =~ ^[0-9]+$ ]]; then
      exit 0;
  else
      exit 1;
  fi
}

est_un_nombre def || echo "Non"

est_un_nombre 12 && echo "Oui"
```

```
$ ./eun.sh; echo $?
??
```

??

Une fonction s’exécute pour ses effets et pour le *exit status* qu’elle retourne avec `return`

Plus efficace qu’une commande : conservée en mémoire et ne crée pas de processus

```
$ cat eun.sh
#!/bin/sh -
est_un_nombre() {
  if [[ $1 =~ ^[0-9]+$ ]]; then
      exit 0;
  else
      exit 1;
  fi
}

est_un_nombre def || echo "Non"

est_un_nombre 12 && echo "Oui"
```

```
$ ./eun.sh; echo $?
1
```

Note : `exit` termine le script, pas juste la fonction ! Donc : seul le premier appel à `est_un_nombre` est effectué, `echo "Non"` n’est jamais appelé et le statut `1` émis par «`echo $!`» du *shell* est celui du `exit 1` de la branche `else` !

Une fonction s’exécute pour ses effets et pour le *exit status* qu’elle retourne avec `return`

Version correcte avec `return` !

```
$ cat eun.sh
#!/bin/sh -
est_un_nombre() {
  if [[ $1 =~ ^[0-9]+$ ]]; then
      return 0;
  else
      return 1;
  fi
}

est_un_nombre def || echo "Non"

est_un_nombre 12 && echo "Oui"
```

```
$ ./eun.sh; echo $?
Non
Oui
0
```

Note : Le *exit status* d’une fonction = celui de la dernière commande exécutée. Cette commande peut être `return` (argument entier).
Quant à celui au niveau du *shell*, ici il est produit par `echo "Oui"`.

Une fonction a un effet durable sur l’environnement, contrairement à un script

Le lancement d’un script crée automatiquement un processus enfant, distinct du processus parent

```
$ cat cd0.sh
#!/bin/sh -
cd "$1"
pwd

$ pwd
/home/tremblay/INF600A/Programmes/Bash

$ ./cd0.sh ~
/home/tremblay

$ pwd
??
```

Une fonction a un effet durable sur l'environnement, contrairement à un script

Le lancement d'un script crée automatiquement un processus enfant, distinct du processus parent

```
$ cat cd0.sh
#!/bin/sh -
cd "$1"
pwd

$ pwd
/home/tremblay/INF600A/Programmes/Bash

$ ./cd0.sh ~
/home/tremblay

$ pwd
/home/tremblay/INF600A/Programmes/Bash
```

Une fonction a un effet durable sur l'environnement, contrairement à un script

L'appel à une fonction ne crée pas un processus enfant

```
$ cdf() { cd "$1"; pwd; }

$ pwd
/home/tremblay/INF600A/Programmes/Bash

$ cdf ~
/home/tremblay

$ pwd
??
```

Une fonction a un effet durable sur l'environnement, contrairement à un script

L'appel à une fonction ne crée pas un processus enfant

```
$ cdf() { cd "$1"; pwd; }

$ pwd
/home/tremblay/INF600A/Programmes/Bash

$ cdf ~
/home/tremblay

$ pwd
/home/tremblay
```

Une fonction a un effet durable sur l'environnement, donc sur les variables, contrairement à un script


```
$ export X=99

$ cat cx.sh
#!/bin/sh -
X=10

$ ./cx.sh

$ echo $X
99
```

```
$ export X=99

$ cx() { X=10; }



$ cx

$ echo $X
10
```

Une fonction a un effet durable sur l’environnement, mais pas sur les variables définies comme `local`


```
$ X=10

$ foo() { X=999; }

$ foo

$ echo $X
999
```

```
$ X=10

$ foo() { local X=999; }

$ foo

$ echo $X
10
```

Pour définir plusieurs fonctions, on les met dans un fichier et on le charge au démarrage

Commande « `.` » = `source`

```
$ ls  -1 ~/cmd/fonctions
cours.f
goto.f
...
terminal.f


$ cat ~/.bashrc
...
for f in $( ls ~/cmd/fonctions/*.f )
do
  source $f
done
...
```

On peut aussi utiliser le mot-clé `function`

Par contre, si on indique `function`, on peut omettre les parenthèses

```
$ function foo { echo "foo"; }

$ type foo
foo is a function
foo ()
{
    echo "foo"
}

$ foo
foo
```

Note : Divers auteurs suggèrent d'utiliser «`foo()`» plutôt que «`function foo`» parce que ce serait plus portable sur différents *shells*. Certains auteurs indiquent aussi qu'on ne devrait pas utiliser `function` en même temps que les parenthèses — l'un ou l'autre, mais pas les deux.

Attention : Si on met la définition sur une seule ligne, le « ; » est crucial !


```
$ foo() { echo "foo" }
> }

$ type foo
foo is a function
foo ()
{
    echo "foo" }
}

$ foo
??
```

Attention : Si on met la définition sur une seule ligne, le « ; » est crucial !


```
$ foo() { echo "foo" }
> }

$ type foo
foo is a function
foo ()
{
    echo "foo" }
}

$ foo
foo }
```

Le nom de la fonction est associé à la variable $FUNCNAME

La variable $0 indique toujours le nom du fichier

```
$ cat foo.sh
#!/bin/bash -

foo() {
    echo $0
    echo $FUNCNAME
    echo $1
}

echo $0
echo $1
echo

foo $1$1
```

```
$ ./foo.sh abc
./foo.sh
abc

./foo.sh
foo
abcabc
```

6. Autres éléments

6.1 Pour déboguer des scripts

On peut faire vérifier un script sans exécuter ses actions avec «bash -n»


```
$ touch foo.foo

$ cat err.sh
#!/bin/sh -
rm -f $1.$1
if ls | grep $1.$1 then
  echo "** Erreur: $1.$1 existe!"
fi


$ bash -n err.sh foo
err.sh: ligne6: Erreur de syntaxe près du symbole inattendu 'f
err.sh: ligne6: `fi'

$ ls foo.foo
foo.foo
```

On peut faire tracer de façon détaillée l'exécution d'un script avec «bash -v»

Trace chaque ligne du script

```
$ touch foo.foo

$ cat err.sh
#!/bin/sh -
rm -f $1.$1
if ls | grep $1.$1; then
  echo "** Erreur: $1.$1 existe!"
fi


$ bash -v err.sh foo
#!/bin/sh -
rm -f $1.$1
if ls | grep $1.$1; then
  echo "** Erreur: $1.$1 existe!"
fi

$ ls foo.foo
ls: impossible d'accéder à foo.foo: Aucun fichier ou dossier de
```

On peut faire tracer de façon détaillée l'exécution d'un script avec «bash -x»

Trace chaque inscription exécutée après expansion

```
$ touch foo.foo

$ cat err.sh
#!/bin/sh -
rm -f $1.$1
if ls | grep $1.$1 then
  echo "** Erreur: $1.$1 existe!"
fi


$ bash -x err.sh foo
+ rm -f foo.foo
+ ls
+ grep foo.foo

$ ls foo.foo
ls: impossible d'accéder à foo.foo: Aucun fichier ou dossier de
```

Donc : Permet souvent de comprendre pourquoi un script ne fonctionne pas… à cause des expansions incorrectes ou des guillemets absents.

La commande «`set -x`» peut être utilisée pour activer une trace à un endroit spécifique du script

Exécution sans trace

```
$ cat args-sans-trace.sh
#!/bin/sh -
echo "Nb. arguments = $#"
for x in "$@"; do
    echo "$x"
done

$ ./args-sans-trace.sh 10 "abc def"
Nb. arguments = 2
10
abc def
```

Note : La commande «`set +x`» désactive la trace. On peut donc alterner entre production et non-production de la trace.

La commande «`set -x`» peut être utilisée pour activer une trace à un endroit spécifique du script

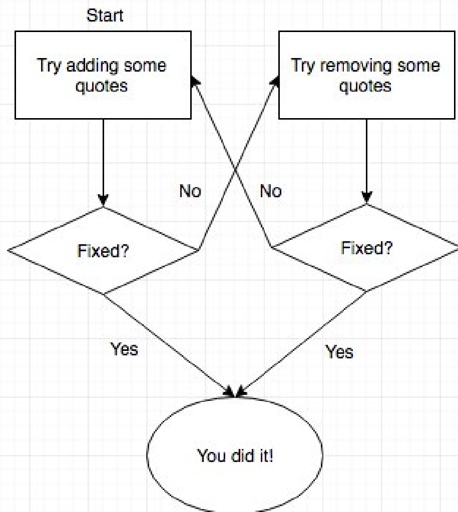
Exécution avec trace : `PS4` définit la chaine affichée devant chaque commande

```
$ cat args-avec-trace.sh
#!/bin/sh -
set -x
PS4="- "
echo "Nb. arguments = $#"
for x in "$@"; do
    echo "$x"
done

$ ./args-avec-trace.sh 10 "abc def"
++ PS4='- '
-- echo 'Nb. arguments = 2'
Nb. arguments = 2
-- for x in '"$@"'
-- echo 10
10
-- for x in '"$@"'
-- echo 'abc def'
abc def
```

En cas de problème, il faut vérifier les guillemets ☺

https://twitter.com/noir/status/735157443014098945



6.2 Exécution et création de processus enfants

Il existe différentes façons de lancer l'exécution de commandes ⋆

Exécution séquentielle

```
$ echo "Bonjour"; echo "le"; echo "monde"
Bonjour
le
monde
```

Il existe différentes façons de lancer l'exécution de commandes ⋆

Exécution concurrente parallèle

```
$ echo "Bonjour" & echo "le" & echo "monde" &
[1] 1280
[2] 1282
Bonjour
[3] 1283
le
$ monde

[1]   Done                      echo "Bonjour"
[2]-  Done                      echo "le"
[3]+  Done                      echo "monde"
$
```

Exécution pipelinée

```
$ echo "monde" | xargs echo "le" | xargs echo "Bonjour"
Bonjour le monde
```

```
$ echo "Bonjour" |\
  xargs -i echo {} "le" |\
  xargs -i echo {} "monde"
Bonjour le monde
```

Il existe différentes façons de lancer l'exécution de commandes

Exécution dans un processus enfant avec attente

```
$ (echo "Bonjour"; echo "le"); echo "monde"
Bonjour
le
monde
```

Il existe différentes façons de lancer l’exécution de commandes ⋆

Exécution dans un processus enfant avec attente

```
$ (echo "Bonjour"; echo "le"); echo "monde"
Bonjour
le
monde
```

Équivalent au pseudocode ci-bas

$ *pid* ← *FORK* (echo "Bonjour"; echo "le")
$ *WAIT_FOR pid*
$ echo "monde"

```
Bonjour
le
monde
```

Il existe différentes façons de lancer l'exécution de commandes

Exécution dans un processus enfant sans attente

```
$ (echo "Bonjour"; echo "le")& echo "monde"
[1] 4373
monde
$ Bonjour
le

[1]+  Done ( echo "Bonjour"; echo "le" )
```

Exécution dans un processus enfant sans attente

```
$ (echo "Bonjour"; echo "le")& echo "monde"
[1] 4373
monde
$ Bonjour
le

[1]+  Done ( echo "Bonjour"; echo "le" )
```

Équivalent au pseudocode ci-bas

```
$ FORK (echo "Bonjour"; echo "le")
$ echo "monde"
...
```

La modification d'une variable par un enfant n'a jamais d'effet dans le parent

Exécution dans un *subshell*

```
$ x=0

$ (x=123; echo $x)
123

$ echo $x
0
```

Exécution en arrière-plan

```
$ x=0

$ x=123 &
[1] 5503
$ echo $x
0
[1]+  Done x=123
```

Donc, il faut bien distinguer entre `( ... )` et `(( ... ))` ⋆

Avec `( ... )`

```
$ x=10

$ ( x=x+1; echo $x )
x+1

$ echo $x
10
```

```
$ x=10

$ ( x=$x+1; echo $x )
10+1

$ echo $x
10
```

Avec `(( ... ))`

```
$ x=10

$ (( x=x+1; echo $x ))
??
```

Donc, il faut bien distinguer entre `(...)` et `((...))` ⋆

Avec `(...)`

```
$ x=10

$ ( x=x+1; echo $x )
x+1

$ echo $x
10
```

```
$ x=10

$ ( x=$x+1; echo $x )
10+1

$ echo $x
10
```

Avec `((...))`

```
$ x=10

$ (( x=x+1; echo $x ))
-bash: ((: x=x+1; echo 10 :
erreur de syntaxe : operateur
arithmetique non valable (error
token is "; echo 10 ")
```

Donc, il faut bien distinguer entre `(...)` et `((...))` ⋆

Avec `((...))`

```
$ x=10

$ (( x=x+1 )); echo $x
11
```

6.3 Les *here documents*

Un *here document* permet de spécifier directement une grande quantité de lignes de texte (*inline*) ★

Forme «ordinaire» : Le terminateur doit être en début de ligne et les expansions sont effectuées

```
$ cat hd1.sh
#!/bin/sh -

cat <<FIN
  Appel a $0 avec "$*"
  FIN
FIN


$ ./hd1.sh abc 10 20 xx
  Appel a ./hd1.sh avec "abc 10 20 xx"
  FIN
```

Un *here document* permet de spécifier directement une grande quantité de lignes de texte (*inline*) ⋆

Forme «avec apostrophes» : Le terminateur doit être en début de ligne et aucune expansion n'est effectuée

```
$ cat hd2.sh
#!/bin/sh -

cat <<'FIN'
  Appel a $0 avec "$*"
  FIN
FIN

$ ./hd2.sh abc 10 20 xx
  Appel a $0 avec "$*"
  FIN
```

Un *here document* permet de spécifier directement une grande quantité de lignes de texte (*inline*)

Forme «avec tiret» : Le texte et le terminateur doivent être indentés par des tabulations

```
$ cat hd3.sh
#!/bin/sh -

cat <<-FIN
	Appel a $0 avec "$*"
	FIN

$ ./hd3.sh abc 10 20 xx
Appel a ./hd3.sh avec "abc 10 20 xx"
```

Note : Et les tabulations en début de ligne sont supprimées lorsque les lignes sont émises !

6.4 La redirection avec «<(...)»

On veut comparer le contenu de deux fichiers après qu’ils aient été triés

```
$ sort foo.txt >foo.trie

$ sort bar.txt >bar.trie

$ diff foo.trie bar.trie
...

$ rm -f {foo,bar}.trie
```

On veut comparer le contenu de deux fichiers après qu’ils aient été triés

```
$ diff <(sort foo.txt) <(sort bar.txt)
...
```

6.5 L’instruction `read`

L'instruction `read` lit sur le flux d'entrée standard et affecte la valeur lue à une variable

Les caractères en rouge sont tapés par l'usager au clavier

```
$ printf "Nom et prenom: "; read nom prenom
Nom et prenom: Tremblay Guy

$ echo $nom
Tremblay

$ echo $prenom
Guy
```

L'instruction `read` lit sur le flux d'entrée standard et affecte la valeur lue à une variable

Les caractères en rouge sont tapés par l'usager au clavier

```
$ printf "Nom et prenom: "; read nom
Nom et prenom: Tremblay Guy

$ echo $nom
Tremblay Guy
```

Note : «le premier mot de cette ligne est affecté au premier nom, le second mot au second nom, et ainsi de suite. Les mots restants sont affectés au dernier nom. S'il y a moins de mots lus dans le flux d'entrée que de variables, celles restantes sont remplies avec des valeurs vides.»
Source : `man read`

L’instruction `read` est parfois utilisée, avec `while`, pour traiter les lignes d’un fichier


```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
tcpdump:x:72:72::/:/sbin/nologin
tremblay:x:1000:1000:tremblay:/home/tremblay:/bin/bash


$ while IFS=: read user pass uid autres; do
> echo $user: $uid
> done < /etc/passwd
root: 0
bin: 1
...
tcpdump: 72
tremblay: 1000
```

L'instruction `read` est parfois utilisée, avec `while`, pour traiter les lignes d'un fichier (bis) ⋆


```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
...
tcpdump:x:72:72::/:/sbin/nologin
tremblay:x:1000:1000:tremblay:/home/tremblay:/bin/bash


$ cat /etc/passwd | while IFS=: read user pass uid autres; do
> echo $user: $uid
> done
root: 0
bin: 1
...
tcpdump: 72
tremblay: 1000
```

L'instruction `read` est parfois utilisée, avec `while`, pour traiter les lignes d'un fichier (ter)


```
$ cat foo.txt
10 20 30
abc def ghi
xx yy zz

$ cat foo.txt  | while read ligne; do
> echo ligne = "$ligne"
> done
ligne = 10 20 30
ligne = abc def ghi
ligne = xx yy zz
```

Notes :

- Exemple utile pour le labo #3.
- Pourrait peut-être (?) aussi être utile à un endroit dans le devoir #1… mais, à utiliser de façon parcimonieuse. Utilisez plutôt `sed`, `grep`, `awk` lorsque c'est possible !

6.6 Suggestions pour le style et la sécurité

Quelques suggestions provenant de deux sources — parmi de nombreuses autres !

> *Unfortunately, Bash is not a language where the correct way to do something is also the easiest. Because it isn't (which is to say that Bash is an unsafe language), it takes discipline to avoid creating those systematic bugs encouraged by the language.*
>
> **Source**: «*Safe ways to do things in bash*»

1. «*Better Bash Scripting in 15 Minutes*»
   `http://robertmuth.blogspot.com/2012/08/better-bash-scripting-in-15-minutes.html`
2. «*Safe ways to do things in bash*»
   `https://github.com/anordal/shellharden/blob/master/how_to_do_things_safely_in_bash.md`

N’hésitez pas à utiliser des guillemets

Quote like a maniac ! *An unquoted variable is to be treated as an armed bomb : It explodes upon contact with whitespace and wildcards.*

Source*:* «Better Bash Scripting in 15 Minutes»

N’hésitez pas à utiliser des guillemets

Sans guillemets

```
$ cat qq0.sh
#!/bin/bash

ch1=$1
ch2=$2

if [ $ch1 = $ch2 ]; then
    echo "Egales"
else
    echo "Pas egales"
fi

$ ./qq0.sh "ab cd" "ab cd"
./qq0.sh: ligne 6 :
     [: trop d’arguments
Pas egales
```

Avec guillemets

```
$ cat qq1.sh
#!/bin/bash

ch1=$1
ch2=$2

if [ "$ch1" = "$ch2" ]; then
    echo "Egales"
else
    echo "Pas egales"
fi

$ ./qq1.sh "ab cd" "ab cd"
Egales
```

Note : Mais ce problème aurait été évité avec `[[...]]` !

Ajoutez des vérifications à faire durant l’exécution

[S]tart every bash script with the following prolog :

```
#!/bin/bash
set -o nounset
set -o errexit
```

This will take care of two very common errors :

1. *Referencing undefined variables (which default to "")*
2. *Ignoring failing commands*

Source: «*Better Bash Scripting in 15 Minutes*»

Ajoutez des vérifications à faire durant l'exécution

Sans vérification

```
$ cat secur0.sh
#!/bin/bash -

echo $a
foo
echo "FIN"

$ ./secur0.sh

./secur0.sh: ligne4: foo : commande introuvable
FIN

$
```

Note : La dernière commande `echo "FIN"` s'exécute même si une erreur a été rencontrée au préalable ☹

Ajoutez des vérifications à faire durant l'exécution

Avec vérification des variables

```
$ cat secur1.sh
#!/bin/bash -
set -o nounset

echo $a
foo
echo "FIN"

$ ./secur1.sh
./secur1.sh: ligne4: a :
   variable sans liaison

$
```

Avec vérification des erreurs

```
$ cat secur2.sh
#!/bin/bash -
set -o errexit

echo $a
foo
echo "FIN"

$ ./secur2.sh

./secur2.sh: ligne5: foo :
   commande introuvable

$
```

Note : La dernière commande `echo` ne s'exécute pas en cas d'erreur préalable ☺

Définissez des fonctions pour faciliter la lecture

Bash lets you define functions which behave like other commands — use them liberally ; it will give your bash scripts a much needed boost in readability [. . .]

Source*:* «Better Bash Scripting in 15 Minutes»

Pour les tests, utilisez les crochets doubles plutôt que les simples

Favor `[[ ]]` *(double brackets) over* `[ ]`

`[[ ]]` *avoids problems like unexpected pathname expansion, offers some syntactical improvements, and adds new functionality [. . .]*

Source*:* «Better Bash Scripting in 15 Minutes»

N'utilisez pas les scripts `bash` pour des tâches trop complexes

Signs you should not be using a bash script

- *your script is longer than a few hundred lines of code*
- *you need data structures beyond simple arrays*
- *you have a hard time working around quoting issues*
- *you do a lot of string manipulation*
- *you do not have much need for invoking other programs or pipe-lining them*
- *you worry about performance*

Instead consider scripting languages like Python or Ruby

Source*:* «Better Bash Scripting in 15 Minutes»

C.J. Johnson.
Pro Bash Programming : Scripting the Linux Shell.
Apress, 2009.

C. Newham and B. Rosenblatt.
Learning the bash shell.
O'Reilly, 2005.

A. Robbins and N.H.F. Beebe.
Classic Shell Scripting.
O'Reilly, 2009.